Type inference for variables and functions (slides)

Jens Gustedt

Type inference for variables and functions

ISO/IEC JTC 1/SC 22/WG14 N2923
WG21 P2305

Jens Gustedt

INRIA – Camus

ICube – ICPS
Université de Strasbourg

https://modernc.gforge.inria.fr/

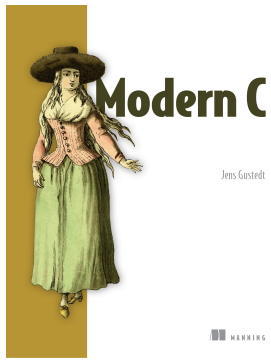

Table of Contents

Policy

- *extend* the standard
  - valid code remains valid
  - new feature integrates syntaxtically and semantically
- fix as much requirements as possible through constraints
  - specific syntax
  - explicit constraints
- avoid new undefined behavior
  - only, if property is not (or hardly) detectable at translation time
  - or we leave design space to implementations
- don't mess with ABI
  - no changes
  - no extensions

Table of Contents

Example: simple function return

How to catch the return of a `<tgmath.h>` function?

```
#include <tgmath.h>
guessType y = cos(x);          // error prone: silent conversion
```

With the new **`typeof`** extension this can already be done:

```
typeof(cos(x)) y = cos(x);   // error prone: paste and copy
```

With **`auto`** this becomes much simpler:

```
auto y = cos(x);                // type safe and copy safe
```

Even works if we don't use `<tgmath.h>`

Example: implementation dependent type

- What is the type of an integer literal? (**int**, **long** or **long long**)

```
[[l]l]div(38484848448, 448484844);
```

- So, which function to use? (**div**, **ldiv**, **lldiv**)
- Use a generic expresssion

```
#define div(X, Y)                \
  _Generix((X)+(Y),              \
           int: div,             \
           long: ldiv,           \
           long long: lldiv)     \
           ((X), (Y))
```

- Which return type? (**div_t**, **ldiv_t**, **lldiv_t**)

```
auto res = div(38484848448, 448484844);
auto a = b * res.quot + res.rem;
```

Example: local variables for macros

```
#define dataCondStoreTG(P, E, D)                    \
  do {                                              \
    auto* _pr_p = (P);                              \
    auto _pr_expected = (E);                        \
    auto _pr_desired = (D);                         \
    bool _pr_c;                                     \
    do {                                            \
      mtx_lock(&_pr_p->mtx);                        \
      _pr_c = (_pr_p->data == _pr_expected);        \
      if (_pr_c) _pr_p->data = _pr_desired;         \
      mtx_unlock(&_pr_p->mtx);                      \
    }  while(!_pr_c);                               \
  } while (false)
```

Table of Contents

Syntax: current

underspecified declarations

- C17's syntax does not impose a type specifier

```
auto y = cos(x); // valid syntax in C17, but constraint violation
```

type completion from initializer

```
double R[] = { 1.0, 2.0 }; // valid syntax in C17, type inferred
```

Syntax: extension

extended use of `auto`

- combination with other storage class specifiers
- allow in file scope
- allow combination with qualifiers including `_Atomic`
- allow combination with pointer

```
static auto const string = "α";
```

Semantic

different qualification and pointer derivation

```
static auto                 const  string1 = "α"; // valid
static auto*                const  string2 = "α"; // valid
static auto                 const* string3 = "α"; // invalid
```

as-if `auto` were replaced by a `typeof`

```
static typeof((0, "α"))       const  string1 = "α";
static typeof((0, "α")[0])*   const  string2 = "α";
static typeof(????)           const* string3 = "α";
```

initializer expression is evaluated exactly once

```
static char*                const  string1 = "α"; // same qual type
static char*                const  string2 = "α"; // same qual type
static XXX                  const* string3 = "α"; // different type
```

Table of Contents

Example: simple function

- The return type is inferred from the **`return`**.

```
// header
inline auto max(time_t a, long b){
  return (a < 0)
         ? ((b <  0) ? ((a < b) ? b : a) : b)
         : ((b >= 0) ? ((a < b) ? b : a) : a);
}
```

- Most of such functions will be **`inline`** or **`static`**.

```
// one TU
auto max(time_t a, long b); // emit external symbol
auto max(time_t, long);     // same
auto max();                 // same
auto max;                   // same
```

- Declaration that is not a definition => only if a definition is visible.

Example: recursive function

- The return type is inferred from the *lexicographic first* **`return`**.

```
inline auto sum(size_t n, strength A[n]){
  switch(n) {
    case 0: return +((strength)0); /* return the promoted type    */
    /* ------------------ sum now visible --------------------    */
    case 1: return +A[0];           /* same type                  */
    default: return sum(n/2, A) + sum(n - n/2, &A[n/2]); /* same */
  }
}
```

Table of Contents

Rules

- Inferred return type must exist before the function definition.
  - Don't use this feature to export types from a function.
- The return type is inferred from the lexicographic first **`return`**.
  - visibility of function name after that
  - visibility of function name after end of body if there is none
  - all other **`return`** expressions must have *same* type
- Use mainly restricted to **`inline`** and **`static`** functions
- Declaration that is not a definition => only if a definition is visible.
  - useful for *instantiation* of **`inline`** functions (objects?)
  - useful for access to file scope identifiers via **`extern`**

Table of Contents

Existing extensions

C++ and widely used gcc extensions

inference	C++	gcc *et al* C extension	current C23
from type	`decltype`		
from lvalue	`decltype`		
from type		`typeof`	`typeof`
from lvalue		`typeof`	`typeof`
from lvalue			`remove_quals`
from value	`auto`	`__auto_type`